IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received October 30, 2021, accepted November 9, 2021, date of publication November 10, 2021,
date of current version November 19, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3127393

GPU-Based Sparse Power Flow Studies With Modified Newton's Method

LEI ZENG, SHADI G. ALAWNEH, (Senior Member, IEEE), AND SEYED ALI AREFIFAR, (Senior Member, IEEE)
Department of Electrical and Computer Engineering, Oakland University, Rochester, MI 48309, USA
Corresponding author: Shadi G. Alawneh (shadialawneh@oakland.edu)

ABSTRACT The Power system is getting larger and more complicated due to development of multiple energy supplies. Solving large-scale power flow equations efficiently plays an essential role in analysis of power system and optimizing their performance during normal or contingencies operation. The traditional Newton-Raphson (NR) algorithm used for power flow calculations is computationally expensive due to updating Jacobian matrix in each iteration. As alternative to update the Jacobian matrix repeatedly, this paper presents a GPU-based sparse modified Newton's method by the introduction of a fixed Jacobian matrix, which integrates vectorization and parallelization technique to accelerate power flow calculations. Moreover, this research in the paper also investigates the performance of the corresponding CPU versions and a MATLAB-based library package, MATPOWER. The comparison of the results on several power system and power distribution systems demonstrate that the GPU variant is more reliable and faster for power flow calculation in large-scale power systems.

INDEX TERMS GPU, CUDA, modified Newton's method, compressed row storage (CRS), Jacobian matrix, vectorization.

I. INTRODUCTION

Power flow studies are one of the most important aspects of power system planning and operation [1]. Nowadays, the power system modeling and analysis have been challenging on power engineers due to the introduction of new energy-supplies and heavier loading, which brings great pressure for power flow calculation [2]. In practice, the conventional NR solver always takes dense matrix format, which consumes much computational resources and storage spaces to calculate power flow. Because of these shortcomings, the traditional NR solver always causes program crash and fail to converge for power systems with over thousands of buses.

The introduction of Graphics Processing Units (GPUs) has brought a revolution in the parallel computing arena [3]. With the benefit of high floating-point processing performance, huge memory bandwidth, and low cost [4], GPUs are not only widely applied to many innovative areas such as artificial intelligence [5], scientific simulation [6]–[9], cryptography [10], [11], integrated circuit analysis [12]–[14], and medical imaging [15], but also many power system applications including optimal power flow [16], power flow [17]–[26], transient stability simulation [27], [28], and contingency analysis [29]. Computer Unified Device Architecture (CUDA), as a general-purpose computing architecture platform, supports many languages including C++, Python, Fortran, OpenCL, OpenMP, and more [30], which makes it easier to explore salient feature of heterogenous computing system at the low-cost of relearning new programming language, eventually achieving dramatic speedups in computing performance [3]. Furthermore, for the benefit of developers working in other languages, NVIDIA CUDA provide us an opportunity to utilize cupy to solve the power flow problem in a vectorization parallelization manner. With the help of these features, one can explore Compressed Row Storage (CRS) matrix format to save more memory and utilize the GPU-based approach to accelerate the power flow calculation for higher efficiency.

GPU-based parallel power flow calculation along with methods to improve the performance has been addressed in literature including [2], [3], [18]–[20], [23], [29], and [31], [32]. Reference [3] suggested that a GPU-based parallel Newton-Raphson's method can achieve a speedup ratio of 3.27 times for a system with 300 buses. Reference [32]

The associate editor coordinating the review of this manuscript and approving it for publication was Hazlie Mokhlis.

presented a speedup of 2.86 times for large power system with GPU-based fast decoupled power flow (FDPF) method. Reference [23] shows that a GPU-based parallel power flow implementation of the conjugate gradient method with Chebyshev preconditioner gives speedup of 10.8 times. Performance comparison involved in GPU-based Gauss-Seidel (GS), Newton-Raphson (NR) and Fast-Decoupled (FD) method are discussed in [20], with speedup ratios of 0.05 times, 1.74 times and 1.30 times, respectively. The GPU-based biconjugate method in [18] presented a speedup of 2.1 times. The GPU-based parallel power flow calculation based Forward-Backward method in [19] gives a speedup ratio of about 1.58 times. From the above findings, GPU-based parallel methods could achieve noticeable acceleration effect in the power flow calculations, compared with sequential executions on a single-core CPU.

Although several papers have been published on GPU-based parallel power flow methods, due to their main focus on dense matrix to leverage the parallel feature of GPUs and improve performance, they can fail to maximize the capability of the GPUs. Specifically, based on the CUDA platform, Wang *et al.* proposed a GPU-based power flow solver with continuous Newton's method [2]. In [2], the method not only needs a great amount of space to store zero elements in admittance and Jacobian matrix, but also consumes much computational resources to calculate many potential nonzero fill-ins in non-coalescing way during the inverting of Jacobian matrix. This can easily cause the programming performance degradation and lead to out-of-memory errors, for the size of Jacobian matrix increases rapidly following power system bus growth. In addition, there also exists a few GPU-based sparse power flow methods such as [33]–[36] and [37]. References [33]–[36] and [37] propose GPU-based parallel sparse lower-upper (LU) factorization method to concentrate on accelerating the solving of linear equations of the power flow, while some with optimized potential operations in [33]–[36], are ignored such as calculating derivatives of the magnitude and angle of bus voltages, creating a fixed Jacobian matrix in a vectorization parallelization manner and so on. In a word, these approaches just only concentrate on optimizing sparse matrix factorization to solve power flow equations and ignore to accelerate the rest parts of power flow calculations. Although the method in [37] considers the whole performance of power flow calculation, dense column major storage in [37] can fail to fully utilize the sparsity of Jacobian matrix in the power systems.

To address such challenges, this paper proposes a GPU-based method with CRS format to save memory for relieving the burden of GPU and CPU and enhance scalability of large-scale power systems. Instead of updating the Jacobian matrix repeatedly and reduce the cost of communication between GPU and CPU in the NR method, a fixed Jacobian matrix is introduced to calculate the unknown state vector such as magnitude and angle of bus voltages. Furthermore, creation of the fixed Jacobian matrix takes the vectorization and parallelization manner, which utilizes coalescing feature of the GPU platform. It can be clear seen that the calculating time of the fixed Jacobian matrix approximately keeps constant despite large-scale power system and improve the performance of the whole power flow calculation. This will be explained in Section IV. The previous work in [2] needs to solely calculate sparse Jacobian matrix inversion, and in this paper, these time-consuming processes are converted into solving of linear equations to avoid the shortcomings of directly inverting large-scale sparse matrix. To further reduce unnecessary memory-consumption and computational overhead, the submatrix of the fixed Jacobian matrix in power system is omitted according to a fast-decoupled power flow (FDPF) method since large-scale power system always has much fewer connections. The GPU-based method in the paper is realized with salient GPU libraries such as cupy. In addition, as for validating the GPU-based method, the corresponding CPU programs are designed. Instead of redesigning new linear equation solvers, the Eigen library with QR solver and LU solver are utilized to solve linear equations and compare with GPU-based solver. This will be further elaborated in Section- V.

The rest of this paper is organized as follows: Section II briefly review the GPU architecture and CRS format to understand the basic concept about GPU and storage of sparse matrices. In Section III, the GPU-based sparse modified Newton's method is proposed based on Runge-Kutta mathematical algorithm. Detailed implementation on the GPU is explained in Section IV. Results and discussion are given in Section V, and Conclusions are shown in the last section.

II. GPU ARCHITECTURE AND CRS FORMAT

In this section, the GPU architecture, a CUDA programming model and CRS format are briefly reviewed.

A. GPU AND CUDA

The philosophy design of GPU architecture is commonly based on throughput-oriented concept which means GPU can invoke thousands of threads to execute simultaneously. CUDA is a suite of technologies, which enable programming on the NVIDA GPUs [34]. An array of streaming multiprocessors (SMs) is the critical hardware in a CUDA-capable GPU. Each SM manages scheduling threads and executes all threads in a warp following the Single Instruction, Multiple Data (SIMD) model. Fig. 1 depicts a typical schematic of a SM consisting of many streaming processors (SP), and each SP contains an arithmetic logical unit (ALU) supporting integer and floating-point arithmetic operations [2]. A register file in each SM has a very short access latency and drastically higher access bandwidth, compared with the global memory.

Fig. 2 describes CUDA programming model called heterogenous programming architecture. The threads lay the foundation of a parallel program and they are organized in the blocks. Furthermore, blocks of threads are grouped into grids, which makes programmers able to explore the capabilities of GPU parallelism. CUDA also assume that both the host (CPU) and the device (GPU) maintain their

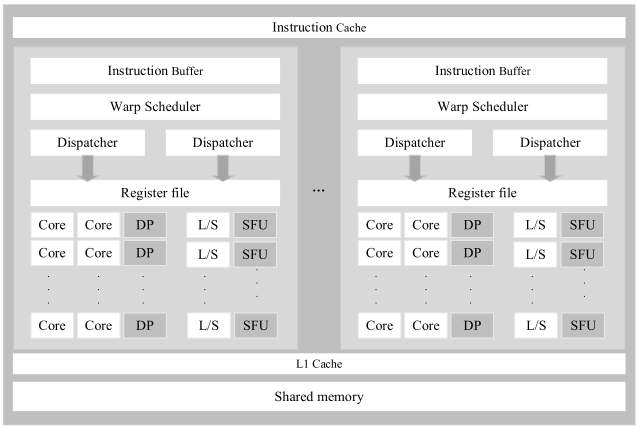


FIGURE 1. A typical architecture of SM.

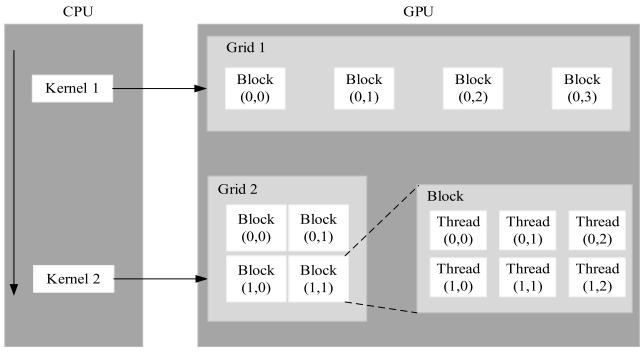


FIGURE 2. The programming model of CUDA.

own separate memory spaces, which are referred to as host memory and device memory respectively [33]. The program flow is controlled by CPU and computationally intensive tasks are offload to GPU by invoking multiple kernels [37]. Specifically, invoking each kernel every time, thousands of threads in the grid are further divided into multiple thread blocks and mapped onto SMs that are referred to as warps. All threads of a warp eventually execute instructions in a lock-step manner in accordance with the Single Program Multiple Data (SPMD) concept [38]. Based on these salient parallel features of NVIDA GPUs, power flow calculation can be operated in a vectorization parallelization manner with cupy.

B. CRS FORMAT

In practice, each bus in a power system has a few links [39]. Thus, it is a good choice to store the admittance and Jacobian matrix in CRS format due to high sparsity of these matrices. Specifically, the CRS format stores the nonzero elements of the matrices in contiguous locations [40]. Unlike storage of dense matrices, the CRS format only stores triple vectors, with respective vectors of nonzero values, column indices and locations. Assuming $A_{n\times n}$ is a general real square sparse matrix. The first vector A_x denotes nonzero vectors in A as the floating-point array and the second vector A_j contains the column indices of the entries in A_x as integers. The last vector A_p contains the location information of A_x. Hence the matrix $A_{n\times n}$ is represented as triplet form:

$$A_{n\times n} = \langle A_x \,|\, A_j \,|\, A_p \rangle \tag{1}$$

From above (1), it was clear that the storage spaces consume less compared with dense matrix format. Instead of storing n^2 elements, just only $2nonzeros + n + 1$ values need to be stored in sparse matrix format [41].

III. GPU-BASED METHOD INTRODUCTION

In this section, the NR method is reviewed, firstly. Then, the modification of NR method based on GPU is introduced.

A. REVIEW OF THE NR METHOD

This subsection briefly presents overview of the NR method and explains the derivation of the method depending on mathematical power flow model. For brevity, all the notion in the paper are presented in Table 1.

TABLE 1. Notion for power system model and matrix operation.

Symbol	Type	Definition
Si	Complex scalar	Complex power at bus i
P_i	Real scalar	Net real power at bus i
Q_i	Real scalar	Net reactive power at bus i
ΔP	Real vector	Real power mismatch
ΔQ	Real vector	Reactive power mismatch
$\Delta\delta$	Real vector	Angle of voltage in state vector
ΔV	Real vector	Magnitude of voltage in state vector
Y_{ik}	Complex scalar	Admittance between bus i and bus k
V_i	Complex scalar	Voltage at bus i
I_i	Complex scalar	Current at bus i
G_{ik}	Real scalar	Conductance between bus i and bus k
B_{ik}	Real scalar	Susceptance between bus i and bus k
θ_{ik}	Real scalar	The difference of voltage difference between bus i and bus k
J	Real Matrix	Jacobian matrix
J_{11}	Real vector	Derivative of ΔP to $\Delta\delta$
J_{12}	Real vector	Derivative of ΔP to ΔV
J_{21}	Real vector	Derivative of ΔQ to $\Delta\delta$
J_{22}	Real vector	Derivative of ΔQ to ΔV
d	abstract	Create a diagonal matrix
∂	abstract	Partial derivatives
V_{norm}	abstract	Normalize the voltage
$concate$	abstract	Concatenate two vectors
Seq	abstract	Generate a continuous integer sequence
$Size$	abstract	Obtain the size of a vector
$flatten$	abstract	Organize data in one dimensional way
abs	abstract	Obtain an absolute value
$conj$	abstract	Conjugate
$transpose$	abstract	Transposition of a matrix
$real$	abstract	Real part of complex vector
$imag$	abstract	Imaginary part of complex vector
csr_mat	abstract	Form a sparse matrix by triplet

The NR method is an iterative algorithm, which is based on linearizing the power flow equations to find the unknown state vector $\Delta\delta$ and ΔV above, firstly, the network of power system should be created. Supposing a power system with n buses, the injection complex power, S_i, at bus i can be defined as [31]:

$$S_i = V_i I_i = V_i(\sum\nolimits_{k=1}^{n} Y_{ik} V_{ik})^* \tag{2}$$

The admittance between bus i and bus j can be further expressed by the conductance G_{ik} and the susceptance B_{ik}:

$$Y_{ik} = |Y_{ik}|\,(cos\theta_{ik} + j\,sin\theta_{ik}) = G_{ik} + jB_{ik} \tag{3}$$

From (2) and (3), the injection complex S_i can be converted to be rectangular form. Then the formulations can be written as:

$$P_i = \sum_{k=1}^{n} |V_i|\,|V_k|\,(G_{ik}\,cos\theta_{ik} + B_{ik}\,sin\,\theta_{ik}) \tag{4}$$

$$Q_i = \sum_{k=1}^{n} |V_i|\,|V_k|\,(G_{ik}\,sin\theta_{ik} - B_{ik}\,cos\,\theta_{ik}) \tag{5}$$

where θ_{ik} is the difference of voltage angles between buses. Since (4) and (5) are a set of nonlinear equations. Taylor series are applied to convert these nonlinear equations to a linear system, which can be expressed in matrix form as:

$$\begin{bmatrix} \Delta P \\ \Delta Q \end{bmatrix} = J \begin{bmatrix} \Delta\delta \\ \Delta V \end{bmatrix} \tag{6}$$

where Jacobian matrix J consists of the partial derivatives of ΔP and ΔQ with respect to the voltage δ and V [1].

$$J = \begin{bmatrix} \frac{\partial \Delta P}{\partial \delta} & \frac{\partial \Delta P}{\partial V} \\ \frac{\partial \Delta Q}{\partial \delta} & \frac{\partial \Delta Q}{\partial V} \end{bmatrix} = \begin{bmatrix} J_{11} & J_{12} \\ J_{21} & J_{22} \end{bmatrix} \tag{7}$$

For a linear system in (6), the NR method is used to evaluate injection power at each bus in the power system and update state vector in each iteration process. Equation (6) is solved iteratively until the tolerance satisfies requirements of the stopping creation.

B. MODIFICATION OF NR METHOD

From above inspiration, the nonlinear power flow are equations are also regarded as:

$$f\,(x) = 0 \tag{8}$$

Equation (8) is expanded with first order Taylor series at the fixed point depend on above NR method. Then, equation (8) can be changed into a set of iterative formulas, and they can be expressed as:

$$x_{n+1} = x_n + \Delta x_n \tag{9}$$

$$\Delta x_n = -J\,(x_n)^{-1} \cdot f\,(x_n) \tag{10}$$

Suppose a set of autonomous ordinary differential equations, as follows:

$$\dot{x} = g\,(x) \tag{11}$$

where $\dot{x} = \frac{dx}{dh}$, equation (11) can be solved for x by integration:

$$\int_{x_n}^{x_{n+1}} dx = \int_{h_n}^{h_{n+1}} g(x)dh \tag{12}$$

Which yields:

$$x_{n+1} - x_n = \int_{h_n}^{h_{n+1}} g(x)dh \tag{13}$$

According to the Euler's method, the integration in (13) can be defined as:

$$\int_{h_n}^{h_{n+1}} g\,(x)\,dh \cong \Delta h \cdot g(x_n) \tag{14}$$

where $\Delta h = h_{n+1} - h_n$, thus, substituting (14) into (13) yields:

$$x_{n+1} = x_n + \Delta x_n \tag{15}$$

$$\Delta x_n = \Delta h \cdot g(x_n) \tag{16}$$

The relationship between (10) and (16) is depicted like (17) if the step size Δh equals one.

$$g(x_n) = -J\,(x_n)^{-1} \cdot f(x_n) \tag{17}$$

because of $\dot{x} = g\,(x)$, the nonlinear equations in (8) can be eventually converted to:

$$\dot{x_n} = -J\,(x_n)^{-1} \cdot f(x_n) \tag{18}$$

Equation (18) is usually considered as the normal continuous Newton's method for power flow [2]. In fact, if derivatives of the state vector in (14) approach to zero, $f\,(x_n)$ should have solutions x_n, as long as $J\,(x_n)$ is not singular. Thus, the Jacobian matrix in the modified GPU-based method does not have to be updated on each iterative process. To reduce the computational overhead, the work in the paper takes a fixed Jacobian matrix J_0, which is usually calculated when x equals x_0 at the beginning of programming. Thus, equation (18) can be further written as:

$$\dot{x_n} = -{J_0}^{-1} \cdot f(x_n) \tag{19}$$

Since the transmission line has very small resistance, the angle of admittance between buses approximately closes to $\pm 90^{\circ}$ and the adjacent buses tend to have a smaller phase angle difference. Thus, J_{12} and J_{21} can be ignored as zero matrices to reduce computational overhead and saves memory space, for the zero elements are not involved in calculation and storage in sparse format. J_0 in (19) can be simplified as:

$$J_0 = \begin{bmatrix} J_{11} & 0 \\ 0 & J_{22} \end{bmatrix} \tag{20}$$

Although above modifications improve the performance, inverting of the Jacobian matrix J_0 solely is still computationally expensive in the processing of (19). Because inverting Jacobian matrix J_0 needs to calculate the many potential fill-ins and allocate much extra memory to store them during inverting of sparse matrix. These steps of inverting the Jacobian matrix J_0 mainly includes three steps, below:

1) Allocate the extra memory for inverse matrix.

2) Implement LU Decomposition and fill the nonzero value in the relative position.

3) Calculate the inverse matrix depending on decomposition in 2).

Steps from 1) to 3) traverse each location of entries in the irregular decomposition matrix on the term-by-term way, which usually introduces many conditional statements such

as if-statements searching for the exact location of the entries. Despite great computation capability of GPU, this way can not only cause the divergence in a warp but also impose a huge scheduling burden on the GPU, especially, when the Jacobian matrix has a huge size. To mitigate the problem, calculation of Jacobian matrix inversion can be converted into solving a set of linear equations. Equations (19) can be viewed as:

$$-J_0\dot{x_n} = f(x_n) \tag{21}$$

Weighing the precision and speed, the second order Runge-Kutta formula is applied to (21); the iterative formulations are defined as:

$$-J_0 \cdot k_1 = f(x_n) \tag{22}$$
$$-J_0 \cdot k_2 = f(x_n + \Delta h \cdot k_1) \tag{23}$$
$$\Delta x_n = \frac{\Delta h}{2} \cdot (k_1 + k_2) \tag{24}$$
$$x_{n+1} = x_n + \Delta x_n \tag{25}$$
$$h_{n+1} = h_n + \Delta h \tag{26}$$

where k_1 and k_2 are intermediate values for updating the state vector Δx_n. Reference [43], [44] suggest that local truncation error and global accumulation error are $O(h^{n+1})$ and $O(h^n)$ respectively. For the viewpoint of speed and precision, the step size Δh is chosen between zero and one, because the error would be smaller as the order increases. Additionally, the step size Δh is also called the fixed time step [2], and the value of Δh can not only determine the iteration number and calculation time, but it also has some extent influence on the convergence of power flow calculation. In general, the iteration numbers and calculation time would decrease as the fixed time step increases. However, the optimal Δh in one power system can probably cause the divergence in the other power system. Weighing the calculation time and convergence of power flow calculation, the compromised value of Δh is chosen to satisfy the all the datasets in the paper. The time step in this method is chosen in the similar way as the factor μ in the reference [2]. Besides, $max\ |\Delta x| < \varepsilon$ is serving as the stop creation in the GPU-based method. Though the method in the paper might increase several iterative times because a fixed Jacobian matrix cannot accelerate the convergence in the process of the power flow calculation. Moreover, the method belongs to the second order numerical methods, which increase the burden of computation. But this high order method is a good choice for GPU, for the fixed Jacobian matrix can reduce the cost of communication between host and device. Especially, in the process of solving (21), steps from 1) to 3) can be omitted, and J_0 just needs transmitting to the device at the beginning of programming. Therefore, these steps in the method can fully utilize the high computing and parallel capability of GPU to compensate the overhead of high order computation. Thanks to the J_0 is a constant, the complexity of high order method can be simplified because the J_0 does not have to be updated following each iteration, compared with traditional NR method. Furthermore, despite J_0 would be involved in calculating state vector on each iteration, the cost of computation is less than that of inverting J_0 directly in the reference [2].

IV. IMPLEMENTATION ON THE GPU

In this section, the heterogeneous architecture and flow chart of GPU-based method are given. Then, creation of a fixed Jacobian matrix in a vectorization and parallelization technique is elaborately discussed.

A. PROGRAMMING ARCHITECTURE

The architecture is a heterogeneous structure, for it consists of CPU and GPU parts. The computationally intensive tasks are offloaded on the GPU, while the rest parts are sequentially executed on the CPU. Both the communication between GPU and CPU is connected by PCI express (PCIe) bus. Hence, the speed of data transmission between them heavily depends on the bandwidth of PCI express. Although the way of transmission has a little negative influence on the performance, the extremely high floating-point processing capability and huge memory bandwidth of GPU would compensate the shortage. In addition, the time-consuming data transmission just have occurred twice in the programming architecture. The first is referred to as loading original datasets, and the other is receiving the results of power flow calculation at the end of programming. This architecture not only reduces computation burden of CPU, but also avoids the cost of repeatedly invoking the GPU's kernel function for transmitting data forward-backward between host and device. With such features, the programming architecture maintains the balance between CPU and GPU and maximizes the acceleration effects, which is depicted by Fig. 3.

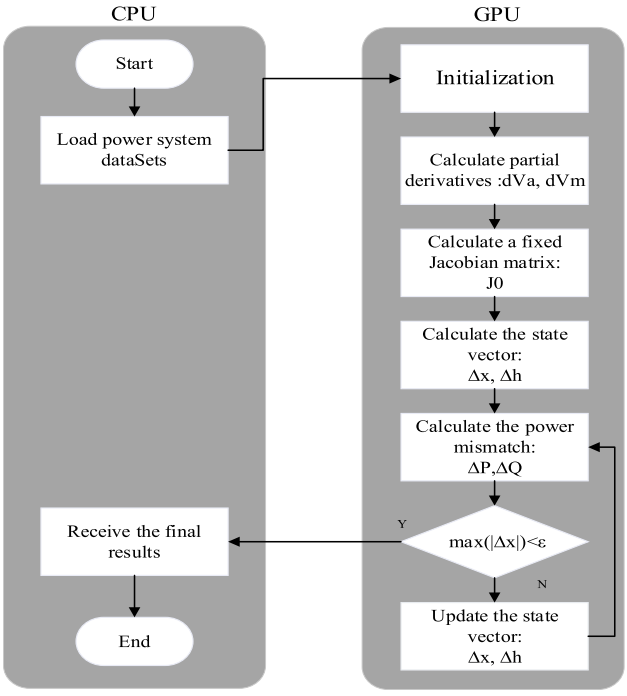


FIGURE 3. The flow chart of programming architecture.

The process of solver in (21) consists of several steps, as follows:

1) load the datasets and initialize them.
2) Calculate partial derivatives of power injection.
3) Calculate a fixed Jacobian matrix in a vectorization manner.
4) Calculate the power mismatch by $f\ (x_n)$ in (21).
5) Evaluate the Δx and Δh.
6) Update the state vector x and h.
7) Check the stop creation.
8) Repeat the process from 4) to 7) until satisfying the stop condition.

B. THE CALCULATION OF JACOBIAN MATRIX

This subsection explains the process of forming a fixed Jacobian in a vectorization manner. This vectorized programs can execute multiple operations concurrently via a single instruction, whereas scalar one can only operate on pairs of operands [50]. Forming a fixed Jacobian matrix consists of two parts including calculating partial derivatives of magnitude and angle of bus voltages, and creation of the Jacobian matrix. Specifically, calculating the partial derivatives such as J_{11} and J_{22} in (20) can be converted to a group of vectorized formulas, as follows:

$$I = Ybus \cdot V \tag{27}$$

$$\partial V_m = d(V_{norm})^* + d(I)^* \cdot d(V_{norm}) \tag{28}$$

$$\partial V_a = 1j \cdot d\ (V) - Ybus \cdot d(V)^* \tag{29}$$

where $Ybus$ is the sparse admittance matrix and unit imaginary number, $1j$.

Algorithm 1 Calculation of Partial Derivatives

```
/* Form a sparse diagonal matrix*/
diag_sparse(v1):
    vec_size = size(v1)
    vec_index = seq(size(v1))
    vec_ptr = seq(size(v1) + 1)
    diag_mat = csr_mat(flatten(v1), flatten(vec_index),
     flatten(vec_ptr))
    return diag_mat
/* Compute the partial derivatives */
dva_dvm(Ybus, v):
    Ibus = Ybus *v
    diagV = diag_sparse(v)
    diagIbus = diag_sparse(Ibus)
    diagVnorm = diag_sparse(v/abs(v))
    dVm = diagV* conj (Ybus* diagVnorm) + conj
    (diagIbus)* diagVnorm
    dVa = 1j* diagV* conj (diagIbus -Ybus* diagV)
    return dVm, dVa
```

In the process of (27), (28) and (29), instead of accessing data in a random manner, all threads in a warp executed the same instruction at any timing point because the entries in the sparse matrix is stored in a consecutive location.

In the case, the technique is usually referred to as coalesces, which is leveraged by the GPU and integrated into cupy libraries. The bold terms in the algorithm 1 respectively denote that the corresponding arithmetic operations are operated on GPU in a vectorization parallel mechanism.

Specifically, Fig. 4 depicts the technique mechanism about implementation of the programs in a vectorization manner.

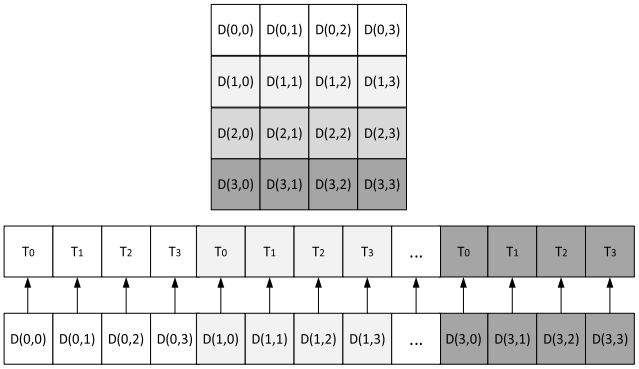


FIGURE 4. The mechanism of vectorization parallelism.

Supposing the sparse admittance matrix $D(4, 4)$ and four threads in a warp, these entries in the matrix have a consecutive location in the memory and are stored in row-major flat mode. During the process, Dynamic random-access memory (DRAM) divides the sequential sixteen locations into four burst sections. All threads in a warp would just cover all entries, while only one DRAM request is executed. Thus, the access is fully coalesced, and no other threads are left. In practice, data from power system can be organized in a coalesced way and the arithmetic operation in Algorithm 1 can be implemented with the benefit of this technique.

The creation of a fixed Jacobian is based on the voltage partial derivative vectors, ∂V_m and ∂V_a. The computational formulas can be written as:

$$J_{11} = real(\partial V_a[pvpq, pvpq]) \tag{30}$$

$$J_{22} = imag(\partial V_a[pq, pq]) \tag{31}$$

where $pvpq$ and pq are respectively corresponding to index vector of PVPQ-buses and PQ-buses. Similarly, the same vectorized idea is also applicable to (30) and (31). However, vectors in $pvpq$ and pq, which contain information of bus indices, are not consecutive. It causes the divergence in a warp and decreases performance because of the irregular storage. To relieve this part, extra memory is allocated to restore the indices. The components in the allocated memory are organized in a consecutive array. Thus, instead of traversing all elements with for-loops on the term-by-term way, the elements in the memory are managed in a coalescing manner as a set of arrays. Despite this approach brings some expense of space, the cost of space is well worth of increasing of speed.

Compared with reference [41], the computational complexity can be reduced from $O(n^2)$ to $O\,(n)$, for the algorithm 2 and algorithm 1 in the paper replaces the traversing with vectorization operation. Although the approach in [41] takes the CRS format to reduce the numbers of iteration, it still needs to traverse all nonzero elements by

Algorithm 2 Creation of a Fixed Jacobian Matrix

```
/* Compute the components of a fixed Jacobian matrix*/
Jaco_sparse(dVm, dVa, pv, pq):
    pvpq = concate(pv,pq)
    J11 = real (dVa[pvpq, transpose(pvpq)])
    J22 = imag (dVm[pq, transpose(pq)])
    Jaco = csr_bmt([J11, None], [None, J22])
    return Jaco
```

two for-loops during the forming Jacobian matrix, which causes a negative influence on performance as power systems increase rapidly.

V. PERFORMANCE RESULT

In the section, firstly, the computing experiment setup and test cases will be introduced. Then, the analysis of storage between sparse format and dense format will be discussed. Next, characteristic curve of forming a fixed Jacobian matrix will be presented. The comparison of overall performance and speed up will be presented. Also, the analysis of stability about CPU-based QR and CPU-based LU method also be discussed. Finally, the distribution networks are tested on the GPU-based method in this paper.

A. TEST PLATFORM AND DATASETS

The computing experiments in the paper are carried out on a workstation equipped with NVIDIA RTX4000 GPU and 8GB DRAM. The workstation has an Intel(R)-i9 CPU and 16 GB RAM. The version of cupy library is 10.2. Thus, the CUDA driver version is also 10.2 in accordance with cupy. The version of MATPOWER and Eigen are 7.1 and 3.4, respectively. The parts of MATPOWER package have been changed into generating a fixed Jacobian matrix instead of dynamic Jacobian matrix. To assure the precision and speed, the stop criterion is set to $1e-4$ for all the power flow. The specification of requirements is shown on Table 2. The test datasets are all from MATPOWER, which can be categorized into two groups as Table 3 and 4.

TABLE 2. Test platform.

Item	Description
CPU	Intel(R) i9 3.10GHz
Memory	16GB RAM
GPU	NVIDIA RTX400
OS	Windows 10
Complier	NVCC
Tool	PyCharm 2021
CUDA	NVIDIA GPU Computing Toolkit 10.2
Library	Eigen 3.4 MATPOWER 7.1 CUPY 10.2

B. ANALYSIS OF STORAGE

For a power system, most spaces are used to store the entries of the admittance matrix and a fixed Jacobian matrix. In this work, the comparison of memory usage between dense matrix format and sparse matrix format is show on the subsection. Since all data in the whole programming are single float-point data type, each element in the matrix would occupy four bytes. In fact, memory is also regarded as a flat model. Therefore, in the analysis of storage, admittance matrix and Jacobian matrix can be combined as whole matrix for simplicity. Therefore, the memory-consumption of admittance matrix and a fixed Jacobian matrix can be cumulated at the same number of power system buses. Table- 5 lists that specific memory-consumption between dense and sparse format. The memory-consumption increases very quickly in dense format. Besides, Fig. 5 explicitly shows that the sparsity of the combined matrix as the numbers of bus increases. It suggests that sparsity would be high when power system has large scale.

TABLE 3. Test case based on IEEE system.

Case	Notion
Case14	Test case form MATPOWE, IEEE system
Case57	Test case form MATPOWE, IEEE system
Case118	Test case form MATPOWE, IEEE system
Case300	Test case form MATPOWE, IEEE system

TABLE 4. Test case based on the other example.

Case	Notion
Case1354	Test case form MATPOWE, Pan-European system
Case2848	Test case form MATPOWE, French system
Case3120	Test case form MATPOWE, Polish system
Case6515	Test case form MATPOWE, French system
Case9241	Test case form MATPOWE, Pan-European system
Case13659	Test case form MATPOWE, Pan-European system

TABLE 5. Memory usage (unit: byte).

Case	Dense	Sparse	Sparsity (%)
Case14	2720	520	80.8824
Case57	57940	2316	96.0028
Case118	186740	4292	97.7016
Case300	1483600	12112	99.1836
Case1354	31284500	51612	99.8350
Case2848	145824320	111744	99.9234
Case3120	182505924	119364	99.9346
Case6515	798285800	265112	99.9668
Case9241	1502485508	415684	99.9723
Case13659	2903875624	561352	99.9807

In addition, Fig. 6 depicts the tendency of memory usage. Considering the above results, memory-consumption almost represents an exponential growth in dense format. Moreover, most of values are zeros due to nature of power system structure. These values not only have no effect on computation but also consume many spaces.

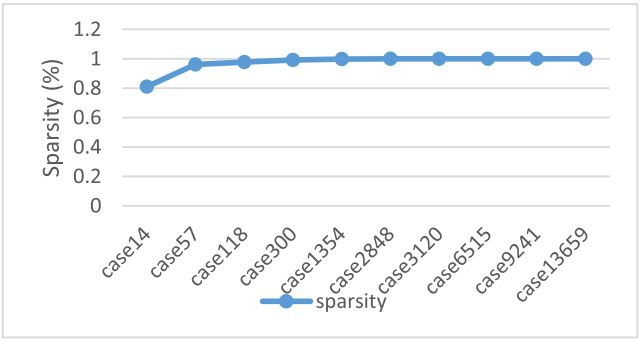


FIGURE 5. The sparsity of the combined matrix.

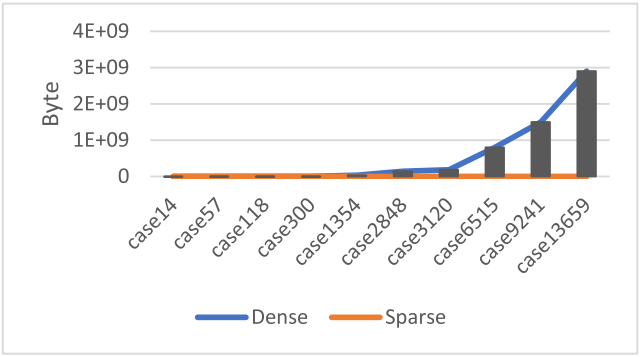


FIGURE 6. The tendency of memory usage respectively in dense and sparse format.

Compared with dense format, memory-consumption in sparse almost keeps a linear increase. The speed of calculation will increase as the storage burden decrease. Moreover, most of components are zeros due to the character of power system. Thus, these zero components not only have no effect on computation but also cause a large waste of memory. Besides, the space complexity can be reduced from $O(n^2)$ to $O(n)$, which provides opportunity to operate a great amount of data for the larger power systems.

C. COMPARSION BEWTEEN CLASSIC AND MODIFIED METHOD

This subsection presents the difference between classic and GPU-based method in the paper. For brevity, IEEE 4 bus system is taken as an example. The speed of classic method on the CPU is faster than the modified method on the GPU because the power system is so small that the GPU takes much time on transmitting Instead of utilizing the high computing capability of GPU. Therefore, the calculation times are 0.6ms and 246.8ms corresponding to the CPU and the GPU, respectively. Table 6 and Table 7 specifically list that calculation results and the norm of power mismatch on the different platform on each iteration. Moreover, Fig. 7, Fig. 8 and Fig. 9 present the voltage magnitude and phase angle profiles based on the classic NR, the FD and the GPU-based method on the IEEE 14 bus system, respectively. Besides, all the Jacobian matrices are taken CRS format in the process of power flow calculation. These profiles of voltage magnitude and phase angle also proved the point that the classic NR method can get convergent solution faster other methods, when the dataset is enough small.

TABLE 6. Execution results on the CPU (unit: per unit).

Iteration number	Norm (power mismatch)	Voltage magnitude ($\|V_m\|$)	Voltage angle ($\|V_a\|$)
		1	0
		0.98972	0.01921
1	0.455209	0.97871	0.03610
		1.01963	0.02764
		1	0
		0.98167	0.01795
2	0.066826	0.96745	0.03315
		1.01964	0.02702
		1	0
		0.98199	0.01664
3	0.016381	0.96810	0.03150
		1.01964	0.02711
		1	0
		0.98230	0.01669
4	0.002514	0.96853	0.03160
		1.01964	0.02711
		1	0
		0.98229	0.01674
5	0.000614	0.96850	0.03166
		1.01964	0.02711
		1	0
		0.98228	0.01674
6	0.000094	0.96849	0.03166
		1.01964	0.02711

TABLE 7. Execution results on the GPU (unit: per unit).

Iteration number	Norm (power mismatch)	Voltage magnitude	Voltage angle
		1	0
		0.99101	0.00910
1	0.171993	0.98416	0.01693
		1.01991	0.01348
		1	0
		0.98660	0.01326
2	0.008173	0.97631	0.02478
		1.01979	0.02027
		1	0
		0.98442	0.15160
3	0.003867	0.97240	0.02844
		1.01973	0.23680
		1	0
		0.98334	0.01603
4	0.001904	0.97045	0.03015
		1.01968	0.02539
		1	0
		0.98281	0.01642
5	0.000953	0.96947	0.03096
		1.01967	0.02625
		1	0
		0.98254	0.01660
6	0.000478	0.96898	0.03133
		1.01965	0.02668
		1	0
		0.98241	0.01667
7	0.000239	0.96873	0.03151
		1.01965	0.02689
		1	0
		0.98234	0.01671
8	0.000120	0.96861	0.03159
		1.01964	0.02700
		1	0
		0.98230	0.01674
9	0.000060	0.96852	0.03166
		1.01964	0.02711

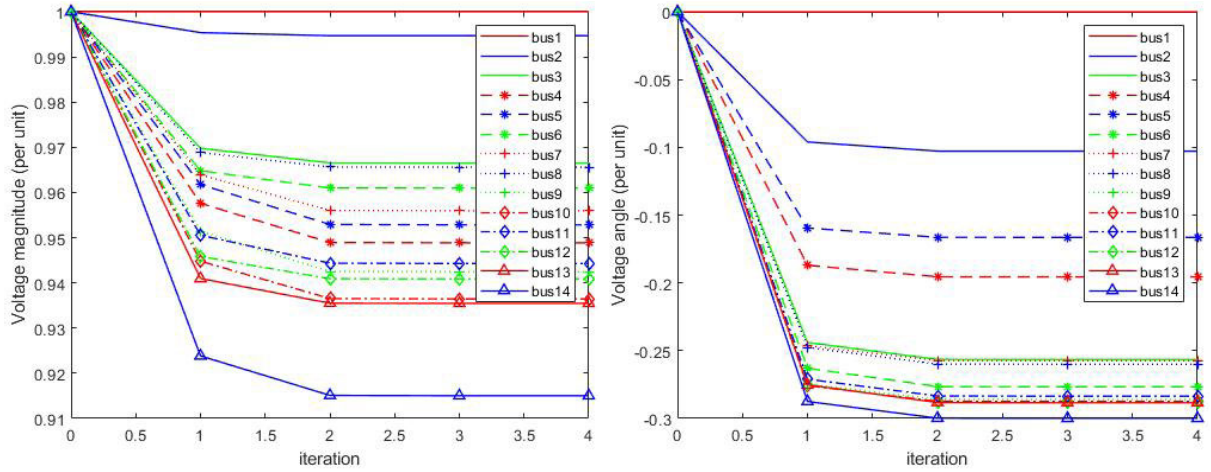


FIGURE 7. Convergence of voltage magnitude and angle for an IEEE bus 14 system based on the NR method.

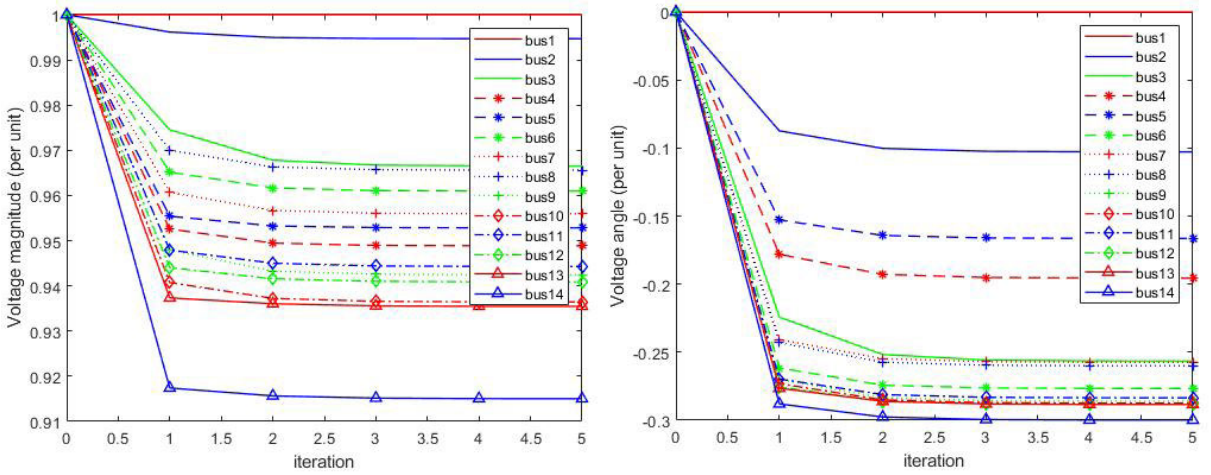


FIGURE 8. Convergence of voltage magnitude and angle for an IEEE bus 14 system based on the FD method.

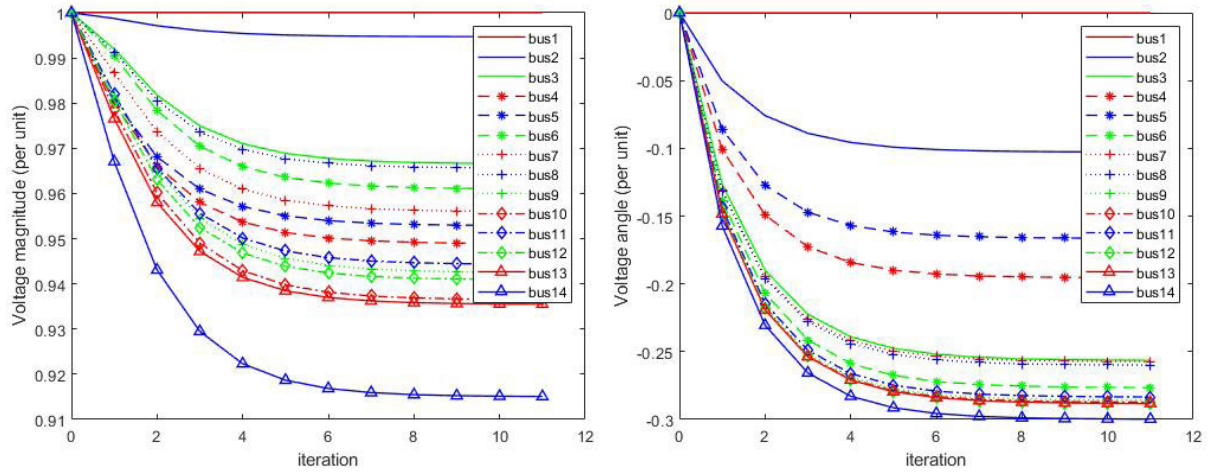


FIGURE 9. Convergence of voltage magnitude and angle for an IEEE bus 14 system based on the GPU-based method.

In addition, Fig. 10 depicts the convergent tendency between the classic NR and the GPU-based method. The classic method has the less iteration if compared with the GPU-based method.

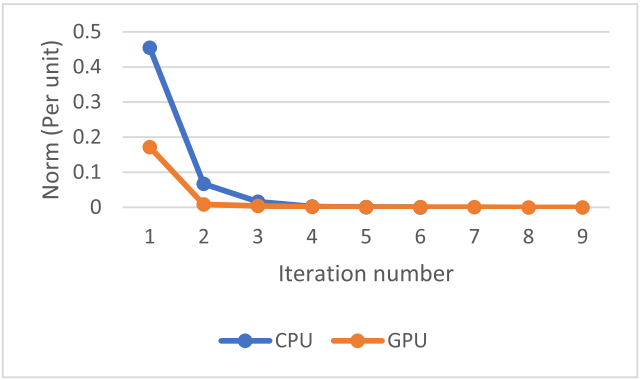


FIGURE 10. Convergence comparison of IEEE bus 4 system between GPU and CPU.

From Table 6 and Table 7, it can be seen that the GPU-based method can work as precisely as the classic method and has more exact value according to the value of norm.

Specifically, the local error of the NR method is $O(h^2)$ because the true solution of (8) is taken a finite number of terms from the Taylor series, and then it can be viewed as:

$$y_{i+1} = y_i + \dot{y}_i \cdot h + O(h^2) \tag{32}$$

where $h = x_{i+1} - x_i$ and $\dot{y}_i$ is the derivatives of power flow equations. For the GPU-based method, the local error is $O(h^3)$ because (8) can be evaluated based on the second Runge-Kutta method:

$$y_{i+1} = y_i + \dot{y}_i \cdot h + \frac{1}{2}\ddot{y}_i \cdot h^2 + O(h^3) \tag{33}$$

Fig. 11 and Fig. 12 simply explain the GPU-based method in this paper can gain more exact values than the classic NR method. Specifically, since the NR method takes the slope at the point x_i to evaluate the value of x_{i+1}, therefore, the value of x_{i+1} should be greater than the true value if $f(x)$ is concave function, as it is depicted by Fig. 8. In addition, the Runge-Kutta method takes the average slope between x_i and x_{i+1} to evaluate the value of x_{i+1}, as is simply described by Fig. 12. Furthermore, (32) and (33) suggest that the computational complexity of the classic NR method is less than the GPU-based method. Therefore, it can provide better performance than the method in this paper, especially when the datasets is small. Although the classic NR method seemed to have an advantage over the GPU-based method when the power system is small, the GPU-based method can provide performance improvement and exact results when the power system is large enough to fully drive parallel potential of GPU.

D. COMPARISION OF GENERATING JACOBIAN MATRIX

This subsection presents results of GPU-based method compared with MATPOWER and CPU variants.

Fig. 13 and Fig. 14 show that the calculating time of creating a fixed Jacobian matrix. For case 9241 and case 13659, the GPU-based method performs better than those of other methods. Most importantly, the calculating time of the method in

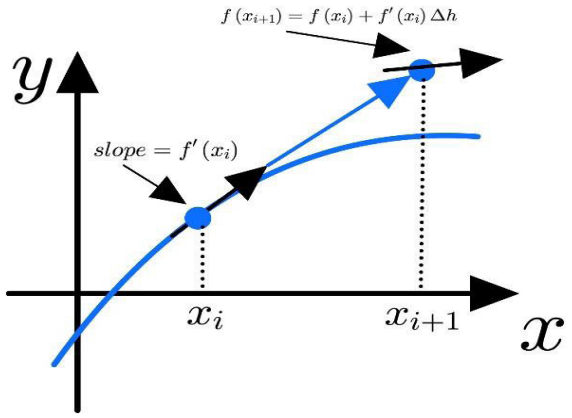


FIGURE 11. An illustration of the original NR method.

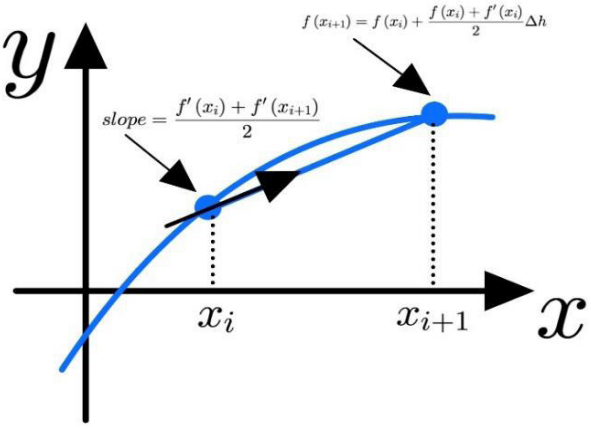


FIGURE 12. An illustration of the GPU-based method.

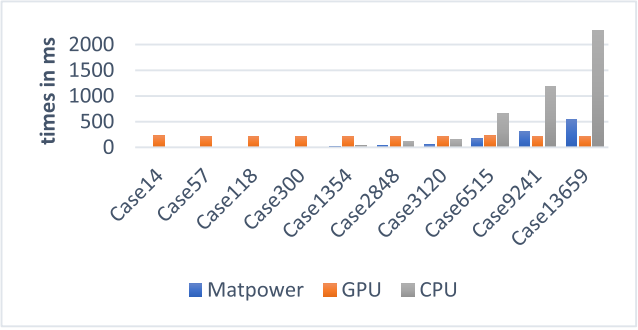


FIGURE 13. The comparison of creating Jacobian matrix.

the paper focuses on data transmission rather than calculations. Thus, the GPU-based method can almost keep constant even if power system is sufficiently large. Although forming the fixed Jacobian matrix, it laid the foundation to accelerate the whole power flow calculation.

On the contrary, MATPOWER and CPU-based methods perform better than that of GPU on small-scale power systems, for they are not involved in communicating between host and device. Once the power system become large enough, computational overhead would degenerate the performance. Therefore, the CPU-based method in Fig. 14

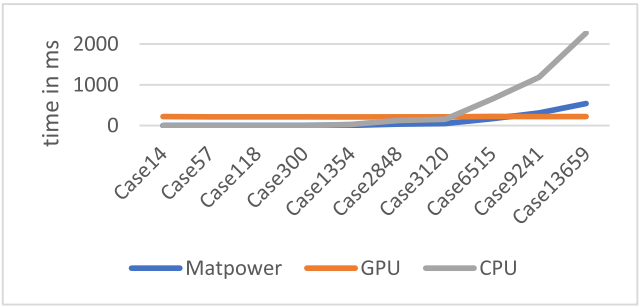


FIGURE 14. The time-consuming tendency of creating Jacobian matrix.

presents a continuous and fast increasing tendency when power system become large. The research building on the above facts demonstrates that the GPU-based method can enhance performance improvement when power systems are large enough to fully drive the parallel potential of GPU. Furthermore, as the power system become larger and more complicated, the GPU-based method can not only maintain a linear computational complexity, but also facilitate acceleration and scalability for the large-scale system in the future.

E. POWER FLOW EQUATIONS SOLVER

In this subsection, the comparison results of different solvers based on different hardware platform will be presented. Two programs of CPU versions are provided in this work, for the CPU-based LU solver is not stable.

Weighing speed and stability, CPU-based LU solver is replaced by the CPU-based QR solver, and the analysis of stability will be discussed.

Table 8 and Table 9 list all implementation results including comparison of calculation time and speedup, respectively, on the GPU and CPU platform. From the results, GPU-based method can achieve outperformance since large systems started around 7000 buses and have an obvious advantage over other methods when the system become sufficiently large.

TABLE 8. Execution results.

Case	MATPOWER (ms)	CPU-LU (ms)	CPU-QR (ms)	GPU (ms)
Case14	7.3	0.3	0.6	251.5
Case57	9.9	2.1	6.3	261
Case118	12.2	3.8	14.2	262.3
Case300	28.7	19.4	819.8	290.6
Case1354	82.9	92.7	43350	351.9
Case2848	226.6	N/A	31416.7	475.9
Case3120	274.5	358	130952.9	522.7
Case6515	523	N/A	178556.2	653.1
Case9241	1141	1946.4	173306.4	941
Case13659	1866.7	3620.7	315310.5	1343.2

For case 13659, the speedup ratios of GPU-based method are respectively 234.8, 2.68, and 1.39 compared with CPU-based QR, CPU-based LU methods and MATPOWR package. In addition, Fig. 15 and Fig. 16 intuitively suggest

TABLE 9. Speedup comparison.

Case	GPU (Baseline)	CPU-LU	CPU-QR	MATPOWER
Case14	1	0.001	0.002	0.029
Case57	1	0.008	0.024	0.038
Case118	1	0.014	0.054	0.047
Case300	1	0.067	2.821	0.099
Case1354	1	0.26	123.2	0.235
Case2848	1	N/A	66.02	0.476
Case3120	1	0.68	250.5	0.525
Case6515	1	N/A	273.4	0.800
Case9241	1	2.07	184.2	1.213
Case13659	1	2.68	234.8	1.390

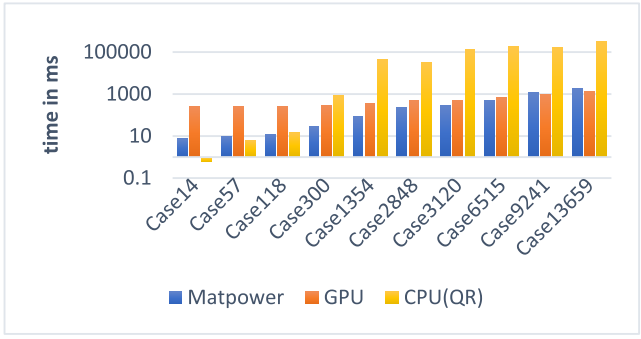


FIGURE 15. Comparison of calculation time (MATPOWER, GPU, CPU-QR).

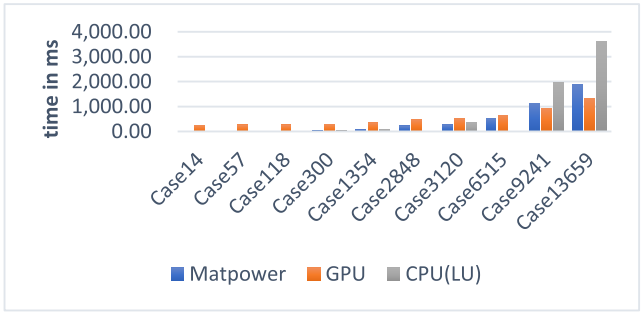


FIGURE 16. Comparison of calculation time (MATPOWER, GPU, CPU-LU).

that GPU-based method also perform better than two other CPU-based methods. However, from Table 8 and Table- 9, it is explicit to suggest stability of CPU-based LU method is the worse one among them. For case 2848 and 6515, the CPU-based LU method can fail to converge. Besides, the Cholesky factorization requires the matrix to be positive definition, which is stricter than the LU factorization. That is why CPU-based QR solver is introduced to replace CPU-base LU solver. In process of LU factorization, LU elimination steps are usually twice as cheap in terms of operations, as QR steps [45], and LU factorization update is based upon matrix-matrix multiplications, where one can use lower triangular matrices [46]. Specifically, the LU factorization is usually depicted, as follows:

$$\begin{bmatrix} l_{11} & 0 \\ l_{21} & L_{22} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} \\ 0 & u_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & A_{22} \end{bmatrix} \quad (34)$$

where $l_{11} = 1$ is a scalar for simplification. All of matrices in (33) are square and portioned identically; this setting leads to a unit lower triangular L. Furthermore, the solving process of (33) can be written as:

$$u_{11} = a_{11} \tag{35}$$
$$u_{12} = a_{12} \tag{36}$$
$$l_{11}u_{11} = a_{21} \tag{37}$$
$$l_{21}u_{12} + L_{22}U_{22} = A_{22} \tag{38}$$

The above process of LU factorization suggests that if the LU factorization exists only if each diagonal entry u_{kk} in (35) is nonzero elements during each LU decomposition. Therefore, for CPU-based LU method, the solver might fail to converge when pivot elements approach zero, because the computer usually regarded these values as unknowns. On the contrary, the QR factorization is always stable, but it requires almost twice as many operations, and a more complicated update step that is not as parallel as a matrix-matrix product [46], [47]. Specifically, the QR factorization is to decompose a matrix with linearly independent columns into a product of an orthogonal matrix [48]. It can be written as:

$$A = Q \times R \tag{39}$$

where A is an $m \times n (m \geq n)$ full column rank matrix, Q is an $m \times m$ orthogonal matrix, and R is an $m \times n$ upper triangular matrix [48]. For brevity, the Householder transformation is employed in A to obtain Q and R. Reference [49] suggests that Q and R can be obtained from Householder Reflection. The formula can be written as:

$$R = H_{n-1} \cdots H_2 H_1 A \tag{40}$$
$$Q = H_1^{-1} H_2^{-1} \cdots H_n^{-1} \tag{41}$$

where H is a set of Householder Reflection matrices. The QR factorization avoids the partial pivoting strategy during the transformation process. Although the method of Householder transformation enhance stability, a set of transformation matrices are generated in QR steps, which causes a huge computational overhead. Fig. 17 just shows that the performance decrease between case 3120 and case 9241 due to the time-consuming operations about matrix decomposition and matrix-matrix multiplication of high order transformation matrices in the process of (40) and (41). Thinking of stability in power system, the tradeoff strategy is that the CPU-based QR solver is introduced as alternative to GPU-based LU solver.

Considering the results and analysis above, the GPU-based method can maintain high speed and stability even if the power systems become sufficiently large, which not only has an overall performance improvement but also provides an opportunity to improve scalability and compatibility for increasing complicated power system in the future.

F. EXTENSION AND TEST OF THE MODIFIED METHOD

This subsection presents comparison of execution results between the GPU-based method in this paper and the GPU-FDPF method in the reference [32]. Furthermore, the GPU-based method is also tested on the distribution networks including IEEE 33bw, IEEE 85 and IEEE 141 bus systems. Table 10 lists power flow calculation of these systems on the GPU-based method.

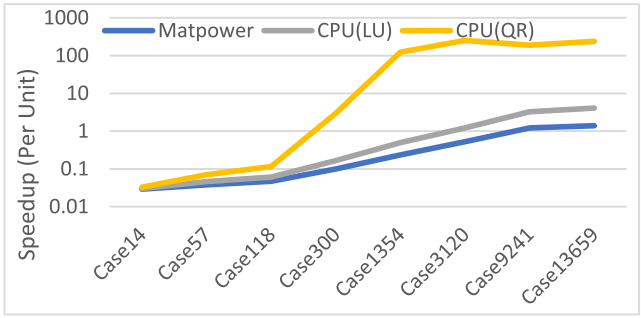


FIGURE 17. Speedup (MATPOWER, CPU-QR, CPU-LU).

TABLE 10. Speedup comparison (CPU: matpower).

Case	GPU-FDPF (Stop criterion: 0.001)	GPU (Stop criterion: 0.001)	GPU (Stop criterion: 0.0001)
Case1354	0.38	0.30	0.24
Case2838	0.28	0.53	0.48
Case3012	0.19	0.55	0.43
Case9241	1.04	1.54	1.21

TABLE 11. Execution results.

Case	Platform	Iteration number	GPU (ms)
Case33bw	GPU	21	139.0
Case85	GPU	24	156.2
Case141	GPU	33	198.3

Taking the fastest CPU-based power flow calculation package, MATPOWER, as a reference, the speedup of the GPU-based method is greater than the GPU-FDPF method in the reference [32]. Aside from case 1354, the GPU-based method performs better than the GPU-FDPF despite the stop criterion is $1e-4$ in the GPU-based method. Although the proposed GPU-based method gives an advantage of almost 1.5 times over the GPU-FDPF method for a system with over 9000 buses, the GPU-based method can provide better performance improvement as the power systems increase in the future.

From the above Table 11, the distribution networks perform better than non-distribution networks. Due to lacking PV nodes, the cost of communication between CPU and GPU is reduced. Therefore, the GPU-based method can fully focus on the high computational capability of the GPU and utilize the multi-threads to optimize the parallelism of the device.

VI. CONCLUSION

This paper presents a GPU-based sparse modified Newton method and demonstrates the accelerated effect for large-scale power systems, which lays the foundation of

large-scale power flow applications. Specifically, the contributions of this work can be summarized as follows:

1) Instead of updating Jacobian matrix on each iteration, a fixed Jacobian matrix is introduced in a vectorization parallelization manner to relieve the computational burden. Although the fixed Jacobian matrix has a negative influence on convergence of power flow calculation, the parameter Δh of Runge-Kutta formula can be adjusted to fix the issue. Besides, the fixed Jacobian matrix might increase iterative numbers, but the cost of communication between host and device is greatly decreased, which computational overhead of iteration can be compensated by the great float-point processing capability of GPU.

In practice, Experiment proves that the time of creating a fixed Jacobian in a vectorization parallelization manner almost maintain a constant time even if the power system is sufficiently large, which improves the efficiency for the whole power flow calculation.

2) Instead of dense matrix format in [2], CRS format is taken to store the elements of the nodal admittance and the Jacobian matrix, which reduces much memory-consumption by excluding storage and computation of zero elements during the process of calculation and consequently, the computational complexity is reduced from $O(n^3)$ to $O(n^2)$.

3) compared with [2], inverting of the sparse Jacobian matrix to update power mismatch is converted into solving linear equations, for inversion of large-scale sparse matrix always not only needs allocate extra memory to store submatrix, but also calculate many potential fill-ins in the inverting process.

The results and analysis in this work demonstrate that the GPU-based method has a great advantage over the version of CPU-based method and MATPOWER, particularly, when power system is sufficiently large. Specifically, The GPU-based modified newton's method, respectively, achieves speedups of 234.8 times, 2.68 times and 1.39 times compared with CPU-based QR, CPU-based LU and MATPOWR, as power system is over 13,000 buses. It also demonstrates better performance results over the GPU-FDPF method in the reference [32]. Furthermore, the GPU-based method is also tested on the distribution networks including IEEE 33bw, IEEE 85 and IEEE 141 bus systems.

In addition, the future improvement can be focused on the GPU-based linear equation solver because creation of Jacobian keeps constant time. Furthermore, based on the analysis of stability, the novel method about GPU-based LU-QR hybrid solvers can be designed to improve performance and stability in the future research, for the hybrid method can combine the advantage of speed of LU method and stability of QR method.

REFERENCES

[1] J. Grainger and W. Stevenson, *Power System Analysis*. New York, NY, USA: McGraw-Hill, 1999, pp. 329–368.
[2] M. Wang, Y. Xia, Y. Chen, and S. Huang, "GPU-based power flow analysis with continuous Newton's method," in *Proc. IEEE Conf. Energy Internet Energy Syst. Integr. (EI2)*, Nov. 2017, pp. 1–5.
[3] J. Singh and I. Aruni, "Accelerating power flow studies on graphics processing unit," in *Proc. Annu. IEEE India Conf. (INDICON)*, Dec. 2010, pp. 1–5.
[4] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," in *Computer Graphics Forum*, vol. 34. Oxford, U.K.: Blackwell, Mar. 2007, pp. 80–113.
[5] K. H. Jin, M. T. McCann, E. Froustey, and M. Unser, "Deep convolutional neural network for inverse problems in imaging," *IEEE Trans. Image Process.*, vol. 26, no. 9, pp. 4509–4522, Sep. 2017.
[6] J. W. H. Liu, "The multifrontal method for sparse matrix solution: Theory and practice," *SIAM Rev.*, vol. 34, no. 1, pp. 82–109, 1992.
[7] *Dense Linear Algebra on GPUs*. Accessed: Mar. 2018. [Online]. Available: https://developer.nvidia.com/cublas
[8] S. Jin, P. Grosset, C. M. Biwer, J. Pulido, J. Tian, D. Tao, and J. Ahrens, "Understanding GPU-based lossy compression for extreme-scale cosmological simulations," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, May 2020, pp. 105–115.
[9] L. Lai, Q. Zhang, H. Tsai, and W.-T. Cheng, "GPU-based hybrid parallel logic simulation for scan patterns," in *Proc. IEEE Int. Test Conf. Asia (ITC-Asia)*, Sep. 2020, pp. 118–123.
[10] W.-K. Lee, R. C.-W. Phan, G.-S. Poh, and B.-M. Goi, "SearchaStore: Fast and secure searchable cloud services," *Cluster Comput.*, vol. 21, pp. 1189–1202, Jul. 2017.
[11] M. Sabbagh, Y. Fei, and D. Kaeli, "A novel GPU overdrive fault attack," in *Proc. 57th ACM/IEEE Design Autom. Conf. (DAC)*, Jul. 2020, pp. 1–6.
[12] V. M. van Santen, F. L. F. Diep, J. Henkel, and H. Amrouch, "Massively parallel circuit setup in GPU-SPICE," *IEEE Trans. Comput.*, early access, Oct. 19, 2020, doi: 10.1109/TC.2020.3032343.
[13] X.-X. Liu, H. Yu, and S. X.-D. Tan, "A GPU-accelerated parallel shooting algorithm for analysis of radio frequency and microwave integrated circuits," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 23, no. 3, pp. 480–492, Mar. 2015.
[14] C. Zhao, Z. Zhou, and D. Wu, "Empyrean ALPS-GT: GPU-accelerated analog circuit simulation," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design*, Nov. 2020, pp. 1–3.
[15] B. D. de Vos, J. M. Wolterink, P. A. de Jong, T. Leiner, M. A. Viergever, and I. Išgum, "ConvNet-based localization of anatomical structures in 3-D medical images," *IEEE Trans. Med. Imag.*, vol. 36, no. 7, pp. 1470–1481, Jul. 2017.
[16] L. Rakai and W. Rosehart, "GPU-accelerated solutions to optimal power flow problems," in *Proc. 47th Hawaii Int. Conf. Syst. Sci.*, Jan. 2014, pp. 2511–2516.
[17] Z.-Q. Wang, S. Wende, V. Berg, and M. Braun, "Fast parallel Newton–Raphson power flow solver for large number of system calculations with CPU and GPU," *CoRR*, vol. 27, pp. 100–483, Sep. 2021.
[18] N. Garcia, "Parallel power flow solutions using a biconjugate gradient algorithm and a Newton method: A GPU-based approach," in *Proc. IEEE PES Gen. Meeting*, Jul. 2010, pp. 1–4.
[19] D. Ablakovic, I. Dzafic, and S. Kecici, "Parallelization of radial three-phase distribution power flow using GPU," in *Proc. 3rd IEEE PES Innov. Smart Grid Technol. Eur. (ISGT Europe)*, Oct. 2012, pp. 1–7.
[20] C. Guo, B. Jiang, H. Yuan, Z. Yang, L. Wang, and S. Ren, "Performance comparisons of parallel power flow solvers on GPU system," in *Proc. IEEE Int. Conf. Embedded Real-Time Comput. Syst. Appl.*, Aug. 2012, pp. 232–239.
[21] X. Li, F. Li, and J. M. Clark, "Exploration of multifrontal method with GPU in power flow computation," in *Proc. IEEE Power Energy Soc. Gen. Meeting*, Jul. 2013, pp. 1–5.
[22] X.-X. Liu, H. Wang, and S. X.-D. Tan, "Parallel power grid analysis using preconditioned GMRES solver on CPU-GPU platforms," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, Nov. 2013, pp. 561–568.
[23] X. Li and F. Li, "GPU-based power flow analysis with Chebyshev preconditioner and conjugate gradient method," *Electr. Power Syst. Res.*, vol. 116, pp. 87–93, Nov. 2014.
[24] X. Li and F. Li, "GPU-based two-step preconditioning for conjugate gradient method in power flow," in *Proc. IEEE Power Energy Soc. Gen. Meeting*, Jul. 2015, pp. 1–5, doi: 10.1109/PESGM.2015.7286544.
[25] H. Cui, F. Li, and X. Fang, "Effective parallelism for equation and Jacobian evaluation in large-scale power flow calculation," *IEEE Trans. Power Syst.*, vol. 36, no. 5, pp. 4872–4875, Sep. 2021.

[26] C. V. Zabala-Oseguera, A. Ramos-Paz, and C. R. Fuerte-Esquivel, "Parallelization of the two-stage state estimation method using GPU-based parallel computing," in *Proc. IEEE Int. Autumn Meeting Power, Electron. Comput. (ROPEC)*, Nov. 2020, pp. 1–6.

[27] V. Jalili-Marandi, Z. Zhou, and V. Dinavahi, "Large-scale transient stability simulation of electrical power systems on parallel GPUs," in *Proc. IEEE Power Energy SoC. Gen. Meeting*, Jul. 2012, pp. 1–11, doi: 10.1109/PESGM.2012.6343968.

[28] Z. Yu, S. Huang, L. Shi, and Y. Chen, "GPU-based JFNG method for power system transient dynamic simulation," in *Proc. Int. Conf. Power Syst. Technol.*, Oct. 2014, pp. 969–975.

[29] A. Gopal, D. Niebur, and S. Venkatasubramanian, "DC power flow based contingency analysis using graphics processing units," in *Proc. IEEE Lausanne Power Tech*, Jul. 2007, pp. 731–736.

[30] D. B. Kirk and W.-M. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, 3rd ed. Cambridge, MA, USA: Elsevier, 2010, pp. 58–59.

[31] D. J. Sooknanan and A. Joshi, "GPU computing using CUDA in the deployment of smart grids," in *Proc. SAI Comput. Conf. (SAI)*, Jul. 2016, pp. 1260–1266.

[32] X. Li, F. Li, H. Yuan, H. Cui, and Q. Hu, "GPU-based fast decoupled power flow with preconditioned iterative solver and inexact Newton method," *IEEE Trans. Power Syst.*, vol. 32, no. 4, pp. 2695–2703, Jul. 2017.

[33] R. Gnanavignesh and U. J. Shenoy, "Parallel sparse LU factorization of power flow Jacobian using GPU," in *Proc. IEEE Region 10 Conf. (TENCON)*, Oct. 2019, pp. 1857–1862.

[34] K. He, S. X.-D. Tan, H. Wang, and G. Shi, "GPU-accelerated parallel sparse LU factorization method for fast circuit analysis," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 24, no. 3, pp. 1140–1150, Mar. 2016.

[35] S. Peng and S. X.-D. Tan, "GLU3.0: Fast GPU-based parallel sparse LU factorization for circuit simulation," *IEEE Des. Test.*, vol. 37, no. 3, pp. 78–90, Jun. 2020.

[36] W.-K. Lee, R. Achar, and M. S. Nakhla, "Dynamic GPU parallel sparse LU factorization for fast circuit simulation," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 26, no. 11, pp. 2518–2529, Nov. 2018.

[37] X. Su, C. He, T. Liu, and L. Wu, "Full parallel power flow solution: A GPU-CPU-based vectorization parallelization and sparse techniques for Newton–Raphson implementation," *IEEE Trans. Smart Grid*, vol. 11, no. 3, pp. 1833–1844, May 2020.

[38] S. Cook, *CUDA Programming: A Developer's Guide to Parallel Computing With GPUs*. Waltham, MA, USA: Morgan Kaufmann, 2012, pp. 84–89.

[39] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Boston, MA, USA: Addison-Wesley, 2010.

[40] M. M. A. Abdelaziz, "OpenCL-accelerated probabilistic power flow for active distribution networks," *IEEE Trans. Sustain. Energy*, vol. 9, no. 3, pp. 1255–1264, Jul. 2018.

[41] F. Schafer and M. Braun, "An efficient open-source implementation to compute the Jacobian matrix for the Newton–Raphson power flow algorithm," in *Proc. IEEE PES Innov. Smart Grid Technol. Conf. Eur. (ISGT-Europe)*, Oct. 2018, pp. 1–6.

[42] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Rommine, and H. van der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, 2nd ed. Philadelphia, PA, USA: SIAM, 1994.

[43] R. L. Burden and J. D. Faires, *Numerical Analysis*. Boston, MA, USA: Brooks/Cole, 2011, pp. 294–296.

[44] M. Faverge, J. Herrmann, J. Langou, B. R. Lowery, Y. Robert, and J. Dongarra, "Designing LU-QR hybrid solvers for performance and stability," in *Proc. IEEE 28th Int. Parallel Distrib. Process. Symp.*, May 2014, pp. 1029–1038.

[45] T. A. Davis, *Direct Methods for Sparse Linear Systems*. Philadelphia, PA, USA: SIAM, 2006, pp. 89–91.

[46] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "A class of parallel tiled linear algebra algorithms for multicore architectures," *Parallel Comput.*, vol. 35, no. 1, pp. 38–53, 2009.

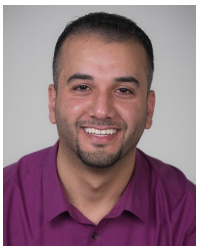
[47] G. Quintana-Ortí, E. S. Quintana-Ortí, R. A. V. D. Geijn, F. G. V. Zee, and E. Chan, "Programming matrix algorithms-by-blocks for thread-level parallelism," *ACM Trans. Math. Softw.*, vol. 36, no. 3, pp. 1–26, Jul. 2009.

[48] C. Luo, K. Zhang, S. Salinas, and P. Li, "SecFact: Secure large-scale QR and LU factorizations," *IEEE Trans. Big Data*, vol. 7, no. 4, pp. 796–807, Oct. 2021, doi: 10.1109/TBDATA.2017.2782809.

[49] I. A. Olajide and M. O. Kolawole, "Examination of QR decomposition and the singular value decomposition methods," *J. Multidisciplinary Eng. Sci. Stud.*, vol. 7, no. 4, pp. 3834–3839, 2021.

[50] (2017). *ArrayFire3.5.1 APIs and Documents*. [Online]. Available: https://arrayfire.org/docs/index.htm

LEI ZENG received the B.S. degree from the College of Food and Bioengineering, Zhengzhou University of Light Industry, Henan, China, in 2014, and the M.S. degree from the College of Electrical and Information Engineering, Zhengzhou University of Light Industry, in 2018. He is currently pursuing the Ph.D. degree with Oakland University, USA. His research interest includes GPU acceleration for computationally intensive applications.



SHADI G. ALAWNEH (Senior Member, IEEE) received the B.E. degree in computer engineering from the Jordan University of Science and Technology, Irbid, Jordan, in 2008, and the M.Eng. and Ph.D. degrees in computer engineering from the Memorial University of Newfoundland, St. John's, NL, Canada, in 2010 and 2014, respectively. Then, he joined the Hardware Acceleration Laboratory, IBM Canada, as a Staff Software Developer, in 2014. After that, he was with C-CORE as a Research Engineer, from 2014 to 2016. He is currently an Assistant Professor with the Department of Electrical and Computer Engineering, Oakland University. He has authored or coauthored scientific publications (including international peer-reviewed journals and conferences). His research interests include parallel and distributed computing, general purpose GPU computing, parallel processing architecture and applications, deep learning, numerical simulation and modeling, automotive applications, and software design.

SEYED ALI AREFIFAR (Senior Member, IEEE) was born in Isfahan, Iran. He received the B.Sc. and M.Sc. degrees (Hons.) in electrical engineering and power systems from the Isfahan University of Technology, Isfahan, in 2001 and 2004, respectively, and the Ph.D. degree in energy systems from the University of Alberta, Edmonton, AB, Canada, in 2010. He was an NSERC Visiting Fellow with the Natural Resources Canada, CanmetENERGY, Varennes-en-Argonne, QC, Canada, from 2011 to 2014. From 2014 to 2016, he was a Postdoctoral Research/Teaching Fellow with the Electrical and Computer Engineering Department, The University of British Columbia, Vancouver, BC, Canada. Since 2016, he has been with the Electrical and Computer Engineering Department, Oakland University, MI, USA, as an Assistant Professor. His current research interests include renewable energies and optimizations in planning and operation of smart grids and microgrids.